NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ON THE INTEGRATION OF LOGIC
PROGRAMMING AND
FUNCTIONAL PROGRAMMING

by

Randy E. Rhodes

June 1985

Thesis Advisor: B. J. Maclennan

Approved for public release; distribution unlimited

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ON THE INTEGRATION OF LOGIC
PROGRAMMING AND
FUNCTIONAL PROGRAMMING

by

Randy E. Rhodes

June 1985

Thesis Advisor: B. J. MacLennan

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*
On The Integration of Logic Programming and Functional Programming

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
June 1985

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Randy E. Rhodes

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, CA 93943

12. REPORT DATE
June 1985

13. NUMBER OF PAGES
71

14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*

15. SECURITY CLASS. *(of this report)*
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*
Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
logic programming, functional programming, integration

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*
Two programming paradigms, logic programming and functional programming, are discussed in detail with emphasis on the particular advantages and disadvantages of each paradigm.
The integration of these two programming paradigms is explored based on the notion that declarative sorts of knowledge (facts and logical relationships) should be expressed in a declarative way, and that procedural sorts of knowledge (manipulation, control, and utilization of knowledge) should be (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

ABSTRACT (Continued)

expressed in a procedural way. Toward this end, the conceptual framework for an integrated language is established, and the basic features of the language are outlined.

Approved for public release; distribution unlimited.

On The Integration of Logic Programming and Functional Programming

by

Randy E. Rhodes
Lieutenant, United States Navy
B.S., Auburn University, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1985

ABSTRACT

Two programming paradigms, logic programming and functional programming, are discussed in detail with emphasis on the particular advantages and disadvantages of each paradigm.

The integration of these two programming paradigms is explored based on the notion that declarative sorts of knowledge (facts and logical relationships) should be expressed in a declarative way, and that procedural sorts of knowledge (manipulation, control, and utilization of knowledge) should be expressed in a procedural way. Toward this end, the conceptual framework for an integrated language is established, and the basic features of the language are outlined.

TABLE OF CONTENTS

I. INTRODUCTION

A. PROGRAMMING LANGUAGE DESIGN

Programming language design represents both an effort to provide the necessary interface with the hardware of the computer and an effort to better capture the ideas of the programmer. As higher order programming languages evolve, a key factor in each language designed is the level of abstraction afforded the programmer. Current conventional languages have removed the programmer from the hardware level of the machine. For instance, instead of being concerned with which registers to use, the programmer can be more concerned with solving the problem at hand. For certain classes of problems, this higher level of abstraction increases the semantic power of the language and better captures the problem solving concepts of the programmer. The evolution of programming language design has resulted in solutions to a broader class of problems and even new approaches toward the solution of presently unsolved problems.

B. PROBLEM COMPLEXITY

The features of the language are the tools with which the programmer tackles a host of complex problems. As the problem complexity increases, the manner in which one works

toward a solution can be affected by the tool or tools available. Consider the analogy of an automobile mechanic working on an automobile engine; a simple tune-up, adjustment, or small part replacement can be performed with simple handtools and devices. However, if the problem is more complex, say involving the cylinders, camshaft, or drive train, then the mechanic cannot solve such problems with simple tools. The solution now requires more advanced tools like hydraulic lifts, pneumatic tools, and precision instruments. In fact, without more advanced tools, the job, if still possible, is solved through improvisation with the simpler tools and results in a less efficient and imprecise solution.

C. SAPIR-WHORF HYPOTHESIS

Similarly, the features of the programming language can effect the manner in which the programmer approaches the solution to a particular problem. This can be considered an application of the controversial Sapir-Whorf hypothesis which originated in linguistic theory [Ref. 1]. Assuming this hypothesis, the programmer attempting to solve complex problems with a limited programming language cannot realize his full problem solving potential and must improvise with the available language features to work toward an acceptable solution.

D. NON-CONVENTIONAL PROGRAMMING LANGUAGES

Conventional programming languages do not offer the programmer a very high level of abstraction, have a restrictive "word-at-a-time" [Ref. 5: p. 404] programming style, and result in what Backus refers to as an "intellectual bottleneck" [Ref. 3]. The sequential nature of these imperative languages, through use of assignment statements to alter memory, results in a von Neumann "mind set", and places limitations upon the level of abstraction available to the programmer. Efforts to provide more semantic power to these languages has resulted in the development of the Ada [Ref. 6] programming language. This very large and very complex language provides increased semantic power at the cost of simplicity, clarity of understanding, and programmer mastery of his tool [Ref. 6].

Non-conventional programming languages, on the other hand, offer a break from the von Neumann mind set and help the programmer approach and solve problems from new perspectives. Such non-conventional programming languages are illustrated by PROLOG [Ref. 2,7], Backus' FP language [Ref. 3], and SMALLTALK [Ref. 4]. Each of these languages represents an implementation of a particular programming language paradigm, namely, logic programming, functional programming (applicative programming with emphasis on functions as arguments), and object-oriented programming, respectively. Issues such as semantics, computational

power, parallelism, side effects, flexibility, simulation, and knowledge representation exemplify some of the basis for development of language design in each of these programming paradigms.

E. RESEARCH FOCUS

With these issues in mind, two programming paradigms, logic programming and functional programming, are discussed in detail. The emphasis of the discussion will be in terms of the particular advantages or disadvantages of each programming paradigm, often exemplified by the PROLOG or "pure" LISP implementation. The focus of this study will be toward the development of a theoretical foundation for the design of an integrated programming language which, of course, maximizes the advantages of both paradigms and minimizes their disadvantages. Such an integrated language should broaden the scope of the programmer's problem solving capability by providing a tool that is both semantically and computationally powerful, and offers improved control characteristics.

II. LOGIC PROGRAMMING

A. BACKGROUND

The foundation for the development of a programming language based upon the rigors of predicate logic seems to have grown out of early attempts to automate theorem proving [Ref. 5, 7, 11] and has subsequently been bolstered by the demands of the artificial intelligence (AI) community in an effort to live up to their rather ambitious name. Hewitt's PLANNER [Ref. 11], a language designed for theorem proving and robot model manipulation, utilized such concepts as backtracking and a database of assertions [Ref. 7], which would later be embraced by the designers of PROLOG. The theoretical foundation for programming in logic, however, is probably best described in Kowalski's work [Ref. 8, 9, 10]. In particular, his paper in the Communications of the ACM [Ref. 8], though concerned with predicate logic as a tool for algorithm analysis, introduces the separation of the logic and control components of an algorithm and strongly suggests the usefulness of this concept in programming languages. Additionally, Kowalski defines the semantics of predicate logic programs [Ref. 9], in a collaboration with van Emden, regarding proof theory and model theory.

A logic program consists of the explicit use of Horn clauses in the process of goal satisfaction. Both Horn clauses and the process of goal satisfaction are described below. Additionally, the advantages and disadvantages afforded the programmer by using logic programming are detailed.

B. HORN CLAUSE

A Horn clause is a subset of the full predicate logic system that is quantifier-free and contains at most one positive literal. It is the preferred logical formula in the expression of logic programs. Horn clauses can be represented by both the logical form, where "-" means negation, and the standard convention, <head> <-- <body>, called a definite clause. The following four classifications of clauses are illustrated by both:

1) Assertion (only one positive literal)

B or B <--

2) Declaration (one positive literal and one or more negative literals)

$B \vee -A_1 \vee \ldots \vee -A_n$ or $B \leftarrow A_1, \ldots, A_n$

3) Denials (no positive literals)

$-A_1 \vee \ldots \vee -A_n$ or $\leftarrow A_1, \ldots, A_n$

4) Contradiction (the empty clause)

$\emptyset$ or <--

C. GOAL SATISFACTION

A logic program consists of a number of these Horn clauses, as axioms, upon which the attempted satisfaction of a particular goal is based [Ref. 12]. An important point here is that these axioms are user defined and basically provide the system interpreter with the facts required to determine whether a given goal is satisfiable.

1. Resolution and Unification

A logic programming interpreter attempts to solve a particular goal statement by resolving any subgoals within the statement with the heads of definite clauses of the logic program. Since the resolution process is one of proof by refutation, a goal is satisfiable if the empty clause (contradiction) can be derived. During the derivation certain bindings for variables may be made which become the solution for the given goal statement [Ref. 12].

There are several algorithms [Ref. 13, 14] for performing such unifications, the foundation of which is described by Kowalski [Ref. 9] and detailed by Hill [Ref. 15], where he names the process LUSH (Linear resolution with Unrestricted Selection for Horn clauses).

The LUSH rule of inference takes a given goal statement $A_1,\ldots,A_n$ and attempts to resolve a subgoal A_i with a definite clause within the logic program that contains an identical form of A_i as the head of the clause. Recall that the actual form of the goal statement is

$-A_1 V...V -A_n$, and that the subgoal being resolved is $-A_i$. Since the head of the clause is the positive literal, the unification of the subgoal and the definite clause resolves the literal A_i, and replaces it with the body of the clause [Ref. 10, 16]. This derivation is possible, of course, only if some general substitution function, mapping variables to terms, makes the subgoal and the head of the definite clause identical [Ref. 10].

2. Non-determinism

Predicate logic is non-deterministic in that the unification process follows a pattern matching scheme for resolution of subgoals and more than one definite clause may have a head that will match [Ref. 10, 12]. Therefore, a resolution procedure like LUSH requires a selection component and a search component [Ref. 16], where the selection component is the rule to determine the search space, and the search component is the strategy whereby that space is searched. This search space can be thought of as a tree with the goal statement as the root and descendant nodes determined by the selection component. The paths of this tree are then traversed according to a strategy given by the search component.

Such a non-deterministic system, then, requires somewhat restrictive selection and search components because of the possibility of an infinite number of paths in the

search space or because of finite paths that do not lead to the empty clause [Ref. 16].

3. PROLOG Example

The selection component of PROLOG provides a rule that always selects the leftmost literal [Ref. 3, 16]. Therefore, the negative literals of a clause must be ordered and fixed, and only those search spaces (or developing trees) can be implemented [Ref. 16]. The search component in PROLOG provides a depth-first strategy with the leftmost-descendant first (derived by the above selection rule). The ordering of descendents is determined by the ordering of the definite clauses within the logic program.

Consider the following logic program (ignoring the structure of the clauses and the unifying substitutions) [Ref. 12]:

(1) A <-- B, C

(2) B <-- D, E

(3) B <-- F

(4) C

(5) D

(6) F <-- G

(7) F

Given the goal statement <-- A, the search space for resolving this goal is depicted as a tree in Figure 2.1 [Ref. 12]. This search tree is an OR-tree whose nodes are possible goals that may occur during resolution of the goal

statement. Each edge of the tree is labeled by the index to the particular clause of the logic program (above) that was used in the resolution process based upon the selection component previously described. The search component progresses through the entire tree until either the empty clause is found or the entire tree is searched [Ref. 12]. Although the unifying substitutions are ignored here, those substitutions, made along a successful path, yield an answer for the goal statement.

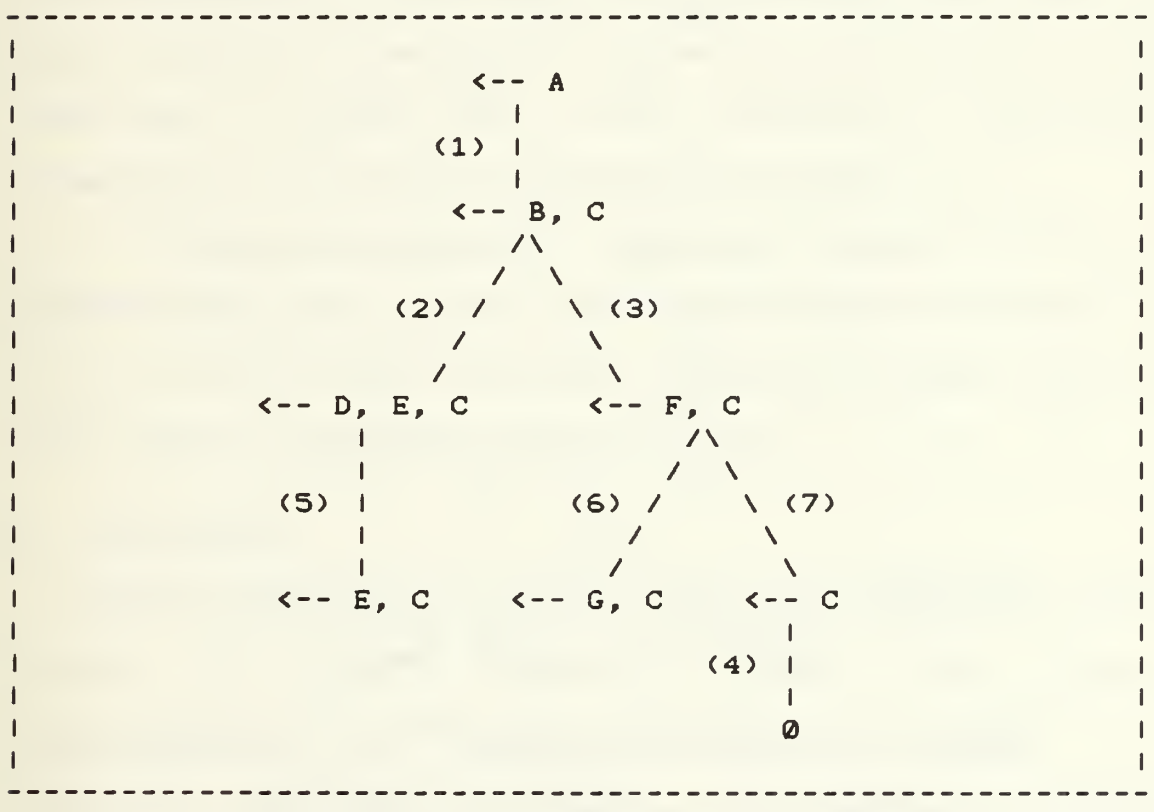

Figure 2.1 Search Space Depicted as a Tree

In order to avoid the redundancy of repeated subgoals in the search tree, many PROLOG systems construct a

proof tree [Ref. 12]. A proof tree is generated for each node in the search tree. It is an AND-tree where the root is the original goal and the leaves are the corresponding subgoals of the search tree [Ref. 12]. The proof trees for nodes <-- D,E,C and <-- F,C are illustrated in figure 2.2

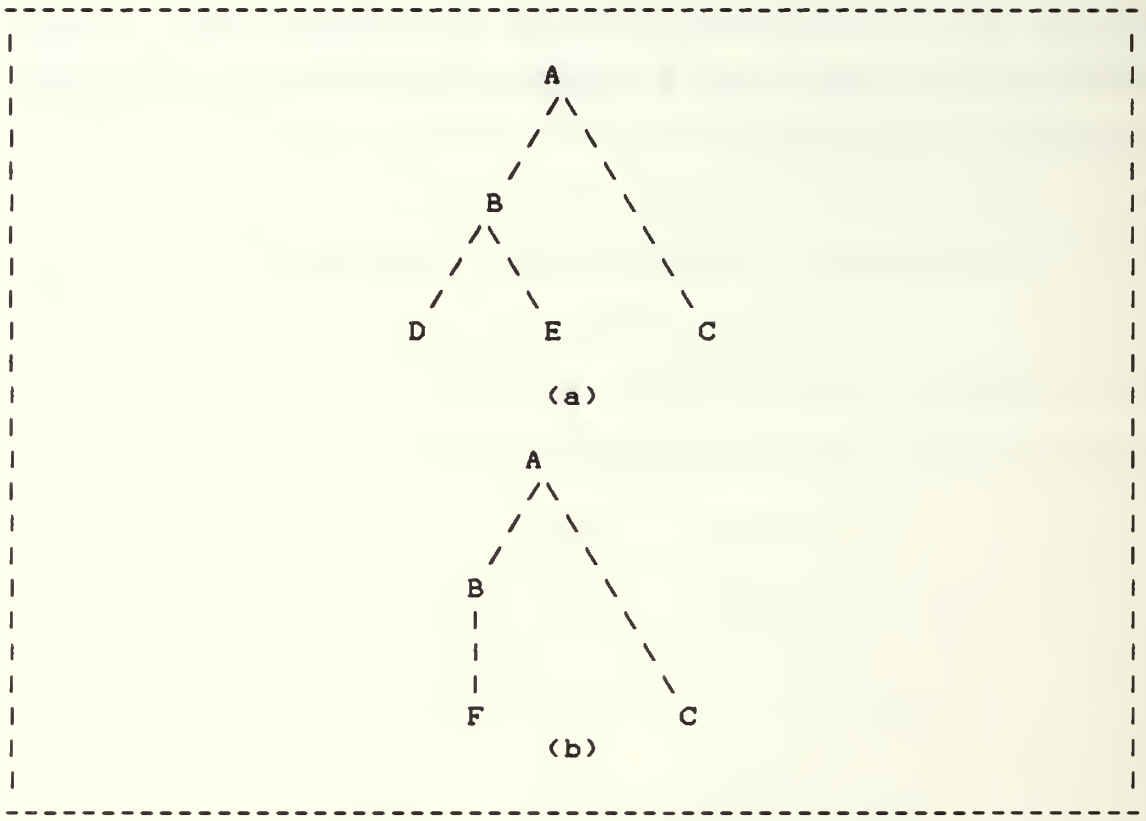

Figure 2.2 Proof Trees

During the resolution process, if a leaf cannot be unified or resolved with the head of another definite clause within the logic program, then a portion of the tree is erased. That portion is from the unresolvable leaf back to the most recent node that has not exhausted its potential

for matching [Ref. 12]. This process is called backtracking and the node at which backtracking stops is called a backtracking point. The search continues from the backtracking point pursuing the unsearched alternatives. A success requires that every path of the proof tree end with the empty clause [Ref. 12].

For example, in Figure 2.2(a), the leaf D can be resolved with the empty clause, but the leaf E cannot, sincethere are no clauses within the logic program that have a head to match it. Therefore the system must backtrack to node B since an alternative choice within the search tree (see label (3), Figure 2.1) is still available. That choice is depicted by the proof tree in Figure 2.2(b).

This backtracking is necessary because of the non-deterministic characteristics of the resolution of subgoals. Yet it is easy to see that fairly large and complex programs would require considerable backtracking.

D. ADVANTAGES

Logic programming offers seductive advantages when dealing with certain classes of problems. Ideas of logic have matured for centuries and have a concise and universally understood semantics. For bodies of knowledge that can be represented in a logical form, logic programming offers a means to prove things about that body of knowledge [Ref. 20].

1. Non-Procedural

The notion of a non-procedural language is one in which the features of the language allow the programmer to concentrate more on "what" the program will do and not on "how" it will be done [Ref. 5: p. 499]. The goal-directed nature of logic programming embodies this notion, in that the programmer expresses the facts, in clause form, which assert the existence of the desired result [Ref. 5: p. 500]. The construction of the desired result, then, is based upon the resolution process of the logic programming language and removes the burden of "how" it will be done from the programmer.

2. Simple Semantics

Much of the power behind the semantics of programming rests with the notions of truth and inference [Ref. 18]. Assertions within the logic program are accepted as truth, and the clauses within the program are facts that allow inferences to be made based upon those assertions.

3. Separation of Logic and Control

Closely related to the non-procedural notion of programming is the notion of a logic component and a control component within the language [Ref. 8]. The control structures of conventional languages determine the order in which actions within the program take place. The fact that statements within that program must be executed in a specified order to ensure correctness illustrates the

interrelationship of those control structures with the actual logic of the program [Ref. 5: p. 510].

Logic programming, on the other hand, allows a much greater separation of logic and control. Since the order of the clauses of a logic program has no effect upon the correctness of the program, the meaning of the program is tied to the logical relationship of the program clauses, not the order in which they are executed [Ref. 2, 5, 8].

This separation of logic and control introduces the notion of separate analysis. Logical analysis is a concern for the correctness of the program, whereas, control analysis is a concern with the efficiency of the program [Ref. 5, 8].

An obvious advantage of this separation, with regard to logic programming, is that the programmer can focus attention upon the details of the logic component when concerned with program correctness. Once a correct program has been established, the programmer can then focus upon the control component for efficiency considerations. This disjoint analysis simplifies the programmer's task by removing the previously dependent relationships between the two components.

E. DISADVANTAGES

Although predicate logic offers some advantages to the programmer in terms of representation of certain kinds of

knowledge, it is important to consider whether predicate logic provides adequate support for reasoning about that knowledge [Ref. 18: p. 139].

1. Undecidability and the Halting Problem

A major drawback of predicate logic is the absence of a decision mechanism for dealing with the knowledge that can be inferred from stated assertions. Without such a mechanism, the resolution procedure blindly searches for a solution.

Pure logic does not allow the expression of heuristics, which hinders the search for a path to a solution during the resolution process [Ref. 19: p. 231]. Therefore the resolution strategy may allow numerous unnecessary and divergent paths to be taken during the search. This can become a grossly inefficient method of search.

Additionally, given a goal statement, using resolution to reason backward will produce a proof if the proposed goal statement is, in fact, a theorem based upon the assertions and clauses in the logic program. However, there is no guarantee that such a search will terminate if there is no proof [Ref. 19: p. 229]. This is a version of the halting problem and is one that the AI community has come to live with in their pursuit of proof methods, being content with a method that proves theorems even if it may

not halt on a non-theorem [Ref. 18: p. 139]. Such procedures are said to be semidecidable.

2. Combinatorial-Explosion

All resolution strategies are subject to the problem of combinatorial-explosion, since the search trees generated can grow very unpredictably [Ref. 19: p. 229]. Somewhat akin to the halting problem, it means that a success for the proof of an actual theorem may be prevented due to the tremendous size and shape of the search space.

3. Axiomatize All Knowledge?

The use of logic programming toward the solution of all problems leads to the restriction that all knowledge associated with the problem must be embodied in axioms [Ref. 19: p. 231]. Such a process might require an enormous effort.

a. Heuristics

A considerable portion of this effort can be attributed to the fact that there is, as mentioned above, no provision for heuristics in the knowledge representation. Therefore, such notions as best, next best, worst, etc. resist representation in logic and make a logical statement of the problem difficult or impossible.

b. No Expression of State

Another drawback of logic programming is the absence of a method for representing state transitions. Without such representation many problems embodied in finite

automata and systems programming become much more difficult to solve.

4. Implementation Considerations

The most notable implementation of logic programming is PROLOG, and the drawbacks noted here are factors that affect efficiency or sacrifice some of the power of the language in favor of more efficient execution.

a. Backtracking and Efficiency

As mentioned earlier, backtracking through a very large search space can be very costly to the search strategy, yet, for such resolution-type systems as PROLOG, it is very necessary. Unfortunately, it is this vast amount of time spent backtracking by the PROLOG interpreter that makes solutions to goal statements very slow in coming.

b. Unification

In order to regain some of this lost efficiency, many PROLOG implementations do not provide full unification. For instance, the resolution process would allow the unification of f(x,x) with f(y, g(y)), and would bind x to g(x) [Ref. 7]. The problem, of course, is that the attempt to prune the search tree allows circularity and the generation of infinite loops.

c. Assertion/Retraction

In the vein of predicate logic problem solving, there have been claims that PROLOG programs have no side effects [Ref. 7]. To some degree this is true, and that

degree rests with those portions of PROLOG which embody the features of predicate logic. For example, the use of 'assert' and 'retract' predicates in the language allows the assertion and retraction of axioms based upon conditions within the logic program.

This violates the principle of predicate logic that each assertion is an independent truth. Therefore, in the resolution process, there are different sets of axioms at different points in time [Ref. 3]. This dependence of some axioms and the addition or deletion of others diverges from the notion of separation of logic and control. In order to provide the programmer with a means to alter a database of facts, the advantages of separation of logic and control (discussed above) are sacrificed.

F. SUMMARY

The logic programming paradigm offers the programmer a high level, non-procedural approach to problem solving enhanced by the simple semantics of Horn clauses. The resolution of goal statements and the unification of variables within that goal is at a level below that of the programmer. Additionally, the inherent capability to separate the logic of the problem solution from the factors which control the solution, allows the programmer to focus attention upon the logical relationships of the problem solution and program correctness.

Disadvantages seem to arise from the fact that all knowledge is not declarative in nature and does not lend itself to axiomatized representation. Furthermore, certain control and efficiency issues are required to curb or contain the search of the knowledge base during the resolution process, and are more naturally represented procedurally.

III. FUNCTIONAL PROGRAMMING

A. BACKGROUND

The foundation of functional programming lies in the notion of general recursive functions, which can express any computable function [Ref. 20: pp. 1-8]. McCarthy's "pure" LISP first illustrated this concept by showing that a number of significant programs could be expressed as pure functions. These pure functions in LISP, of course, operate on list structures, but the notion may be generalized to other structures.

The importance of the recursive function concept is the impact that it has on the nature of programming. In conventional languages imperative statements are used to alter control flow and update memory. These statements (as mentioned in chapter 2) are dependent upon the order in which they are executed. The recursive use of pure functions eliminates the requirement for these imperative statements and, as a result, is often called "assignment-less" or "variable-less" programming. This "value-oriented" program-ming is based upon the use of pure expressions (discussed below) and offers the advantages of arithmetic and algebraic expressions to the programming language [Ref. 20: p. 1-1].

Probably one of the most critical attacks upon the conventional programming languages of the von Neumann style came from Backus' Turing Award Lecture, where he describes them as "fat and flabby" [Ref. 3: p. 614]. His criticism of the framework, the "word-at-a-time" programming, and the lack of useful mathematical properties of conventional programming languages [Ref. 3: p. 617] led to the design of his Functional Programming (FP) System.

An important contribution in Backus' FP paradigm is the emphasis on the use of functionals (described below). The use of functionals allows the programmer to raise himself above the recursive nature of the function by providing a higher level of abstraction. At this higher level, the programs can be made more understandable and thereby much easier to maintain.

B. EXPRESSIONS

Expressions may be arithmetic, relational, or boolean, as illustrated in Figure 3.1. In conventional

arithmetic:	(a + b) * c
relational:	a + b = 0
boolean:	-(a V b)

Figure 3.1 Types of Expressions

languages these types of expressions appear on the right-hand side of an assignment statement. By eliminating the use of the assignment statement we can concern ourselves with "pure" expressions and the properties associated with them [Ref. 27: p. 28]. These properties are listed in Figure 3.2 and several are discussed below.

* value is independent of the evaluation order
* referential transparency
* no side effects
* inputs to an operation are obvious in the written form
* effects of an operation are obvious in the written form

Figure 3.2 Properties of Pure Expressions

1. Evaluation Order Independence

An important property of pure expressions is the fact that within a given context, an expression has the same value regardless of the order in which it is evaluated. In fact, the evaluation of subexpressions within a given expression will not effect the evaluation of other subexpressions, and the order in which they are evaluated will not alter the final value of the overall expression.

This independence of evaluation order (also called the Church-Rosser property [Ref. 20: p. 1-3] is illustrated in Figure 3.3. Here a pure expression (with subexpressions) is shown in tree form, where the evaluation begins at the leaves of the tree. As soon as the leaves below an operator

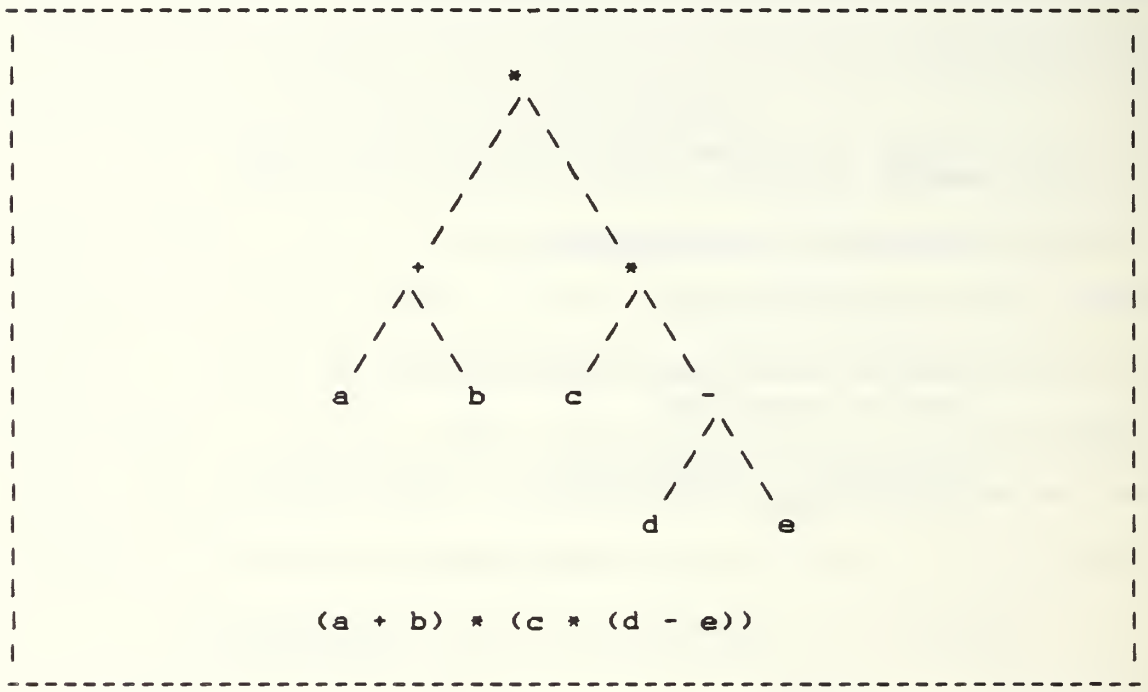

Figure 3.3 Pure Expression as a Tree

node have values, that operator can be applied to those values and that subexpression is evaluated. Once evaluated, those subexpressions, as in this example, may become one of the arguments to another operator. Note that whether the "+" operator or the "-" operator is evaluated first does not alter the value of the entire expression.

The importance of context can be illustrated by an "impure" expression, as in figure 3.4, where assignments to variables can be made. If memory outside the context is allowed to be altered, then the expression is not "pure" and side effects can result. In this case the value of the variable "a" can be altered by the evaluation of the function call.

```
a + b * F(c)

  where  F(z:integer): integer
         begin
           a := 0;
           F = z * z
         end;
```

Figure 3.4 Impure Expression

2. Referential Transparency

The property that the replacement of an expression (or subexpression) by its value is entirely independent of the surrounding expression in which it occurs is called referential transparency [Ref. 20: p. 1-3]. This property means that having evaluated an expression, it need not be evaluated again. This provides the universal ability to substitute equals for equals within a given context.

For example, given the context b=3 and c=4 for the expression in Figure 3.5, referential transparency means

that having evaluated (b + c) to the number 7, the substitution of 7 for the other occurence of (b + c) will not affect the value of the overall expression.

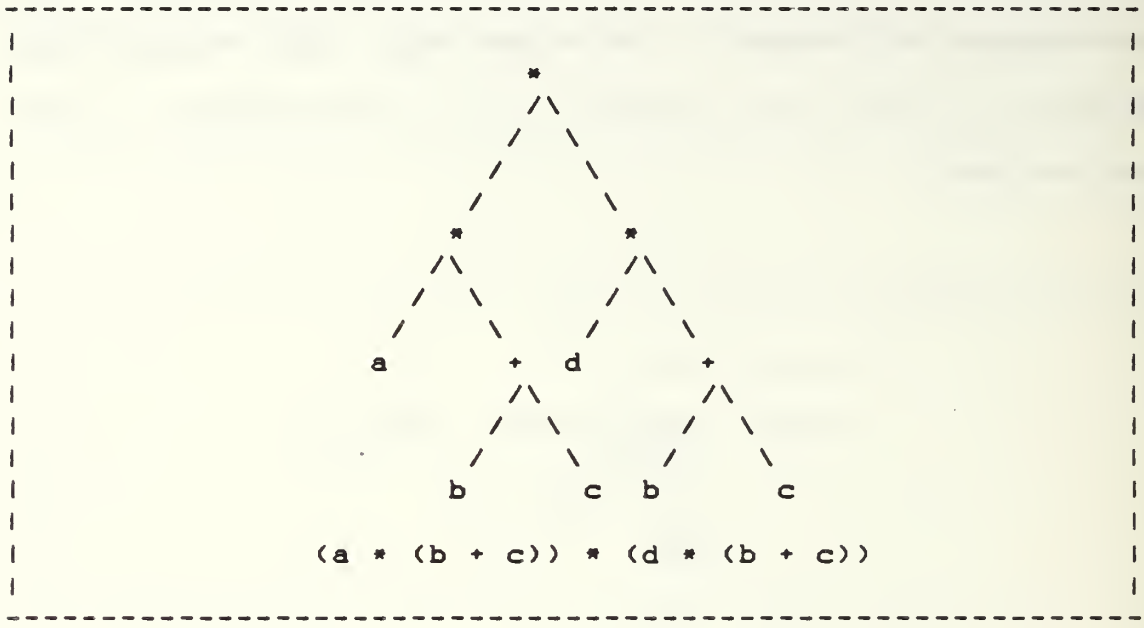

Figure 3.5 Pure Expression as a Tree

C. FUNCTIONS

Mathematical mappings from inputs to outputs are "pure" functions. Such pure functions are the "+", "*", and "-" operators of the pure expression in Figure 3.4. The results of these operations depends only upon the inputs. In fact, the notions of pure expressions and pure functions form an interesting dependency. In order for an expression to be pure (thus having the properties stated above) it must consist of pure functions. Additionally, if functions can be constructed with pure expressions (containing no

explicit or hidden assignment statements), then the function will retain the properties of pure expressions [Ref. 20: p. 1-5].

1. Function Application

An applicative program takes the form of an expression that consists of the application of pure functions to their arguments. Thus function application is the fundamental operation of applicative programming and is illustrated, in Figure 3.6, by the prefix function form of the expression in Figure 3.5.

```
times((times(plus(b,c),a)),(times(plus(b,c),d)))
```

Figure 3.6 Prefix Form

Within the applicative programming language functions may be defined explicitly, conditionally, recursively, or as the composition of other functions. The important point, however, is that these functions operate only on data (characters, numbers, etc.).

2. Functionals

In order to provide a higher level of abstraction, functionals are functions that take other functions as arguments. Functionals result from identifying recurring patterns in function definitions and abstracting them to a

higher level. The "map" functional is an excellent example, since, it accepts functions such as "times", "plus", etc. and maps them onto a list of ordered pairs. Such a functional eliminates the requirement to explicitly define "map_times", "map_plus", etc. functions.

Functional programming, then, is a form of applicative programming that makes extensive use of functionals. Not only does it simplify the programming process (fewer explicitly defined functions), but also offers additional properties which are listed in Figure 3.7 [Ref. 27: p. 30].

* easy to use existing functions to build new ones
* easy to combine functions using composition
* subject to algebraic manipulation
* easier to prove correct
* easier to understand

Figure 3.7 Properties of Functional Programs

D. PROOF OF CORRECTNESS

The mathematical properties of functional programming lend themselves to much more straightforward proof of correctness than either imperative languages or logic-based languages. Most often, the recursive function definitions of the functional program can be individually proved by

induction. Additionally, the functional programs themselves are subject to algebraic manipulation. A detailed analysis of such algebraic properties is presented in Backus' Turing Award Paper [Ref. 3].

A comparison of Hoare's axiomatic model of correctness [Ref. 28] with that of Mill's functional model of correctness [Ref. 29] helps to illustrate this more straightforward proof of correctness method.

1. Hoare's Axiomatic Model

Hoare's axiomatic model of correctness uses the notation

(P) S (Q)

to state the required connection between the input assertion P, output assertion Q, and the program (or part of a program) S. Partial correctness of program S results if and only if for every substitution of values which makes P true, then after execution of S, Q must be true. Total correctness results if it is proven that if P is true then S terminates [Ref. 21]. Hoare's rules of inference, very similar to the rules of predicate logic, are used to prove correctness of particular programs. By assuming the pre- and postassertions of every program statement, as well as the program itself, the rules of inference are used on each piece of the hierarchy to establish the proof. The problems arise from the fact that most statements of a program do not annotate their pre- and postconditions and that the proof of

iterative portions of the program requires recognition of a "loop invariant" that is often difficult to ascertain. [Ref. 21].

2. Mill's Functional Model

Mill's functional model of correctness states the intended function of a program as a functional abstraction which summarizes the outcomes of the program (or part under consideration). This functional abstraction is independent of the control structures and data operations and reduces the question of correctness to one of function composition and function equivalence [Ref. 21]. Partial correctness of program S means that "with respect to function F, every argument X, for which F is defined and F(X)=Y, then if program S is executed with initial input vector X, its final output vector is Y." Total correctness is proven by showing that if X is in the domain of F then S terminates [Ref. 21]. The problem of determination of the loop invariant is minimized since the intended function of the loop may be easily converted to a loop invariant.The problem still remains that in most conventional programming languages most statements of the program do not annotate their function.

E. ADVANTAGES

1. Higher Level of Abstraction

The advantages to be gained by functional programming are somewhat analagous to the advantages of structured

programming. The higher level of abstraction afforded by "goto-less" programming makes it easier to reason about and understand programs. The gotos exist at a lower level of abstraction and the programmer is not burdened with those details. Similarly, the "assignment-less" property of functional programming encourages an even higher level of abstraction, providing a more systematic derivation of programs and resulting in greater understandability [Ref. 20: p. 3-4]. Assignments, of course, exist but are hidden from the higher level of abstraction.

Additionally, the functionals within the language provide a mechanism for achieving an even higher level of abstraction. Common patterns among user-defined functions can be abstracted out, named, and thereafter referred to without concern for the underlying function composition.

2. No Side Effects

Many of the side effects associated with imperative programs result from the assignment statement and its use in altering variables (local and non-local). This results in hidden interfaces within the program, which degrade both program correctness and understandability. In functional programming the assignment statement is eliminated and the interfaces manifest themselves in the expressions of the program. This means that the input-output connections of the subexpressions within an expression are visually obvious

[Ref. 20: p. 1-4] and confer no hidden interfaces or side effects.

3. Verification and Proof Techniques

The functional model has several advantages over the axiomatic approach. By stating specifications and sub-specifications as functions from an input space to an output space, the functional model is a mathematical model in the strictest sense. The axiomatic approach organizes such specifications into Boolean functions represented by assertions on program variables, assertions given in terms of the relationship of the variables involved. The functional approach is in terms of the relationship of the two value sets involved. This means that the axiomatic approach maps from the current values of the variables into the two-tuple [True, False] instead of the more mathematical functional model which maps from the input value space to the output value space. Another advantage is that changes in a program that do not affect another program segment do not require a new proof of correctness for that segment. This results from the fact that the proof of a functional specification is in terms of the behavior of the program statement independent of the history of variables in the segment. The assertions of the axiomatic approach, however, are restricted by variable history and interdependence with other variables [Ref. 21]. Additionally, different implementations of a particular specification can be

substituted without requiring new proofs of other program segments.

Functional programming and the functional model described go hand-in-hand toward meeting the goals of structured programming. The decomposition of the larger programming structure into simpler structures (stepwise refinement) is easily afforded with functional programming in which larger programs or functions are merely compositions of simpler functions. The problem mentioned above regarding conventional languages and how each statement rarely annotates its function is eliminated with functional programming. Therefore, the functional program lends itself to proof of correctness with the discussed model in a convenient manner.

4. Parallelism

The ability to perform parallel execution in functional programming is a direct result of the property of evaluation order independence inherited from pure expressions. The various nonoverlapping subexpressions within an expression can be evaluated simultaneously since the evaluation of one is not dependent upon the evaluation of another. Therefore, a multiprocessor could assign various processors to evaluate different parts of an expression in parallel.

Unlike conventional languages which require the programmer to identify the portions of a program which can

be run concurrently, a functional language can handle as many processors as there are subexpressions to evaluate, and the order in which the processors are assigned, or subexpressions are evaluated will not alter the final evaluation (it may, of course, affect the efficiency of execution).

F. DISADVANTAGES

1. Limited Problem Domain

Although the mathematical properties of functional programming offer advantages, certain tradeoffs do result from those properties. These tradeoffs have a limiting effect upon the problem domain to which functional programming solutions are practical, or even feasible.

Functional programming provides no notion of state nor does it provide any notion of time. This weakness in maintaining temporal relations restricts the use of functional programming for such state-oriented applications as operating systems, database management, or discrete simulation.

2. Recursion and Inefficiency

The recursive function definition is an important component within a functional programming language and is probably the most expensive. The expense of numerous recursive calls can be minimized if the hardware support can take advantage of the parallelism afforded by the language.

Unfortunately, such multiprocessor support for functional languages is not widespread and the use of such a language as "pure" LISP on a uniprocessor can be very slow depending upon the nested levels of recursion. Without the support for parallel execution, efficiency can quickly become a major factor in the effectiveness of the programming language.

3. Industry Resistance to Change

As with most new concepts, the resistance to change surfaces whenever the status quo is threatened. Most of industry is still tied to the von Neumann architecture and "mind set" (both financially and intellectually). Until the decisionmakers within the industrial complex are convinced that the advantages afforded by new concepts will outweigh the expenditure in time, personnel training, and money, these new concepts will remain at the theoretical or experimental level.

G. SUMMARY

The functional programming paradigm provides the programmer with a very high level of abstraction, making it easier to reason about and understand programs. In contrast to von Neumann languages, functional languages are free from side effects resulting from heavy dependance upon the assignment statement. Additionally, the non-sequential nature of functional programming, based upon the property of

evaluation order independence, lends itself to parallel execution in a multi-processor environment.

The disadvantages of the functional programming paradigm rest with its somewhat limited problem domain, because of its weakness in representing temporal relationships. Although functional languages lend themselves to parallel execution, without more effective use, in terms of hardware support, of the parallel nature of the language, the cost of numerous recursive calls is inefficiency.

IV. FEASIBILITY OF AN INTEGRATED LANGUAGE

A. PROCEDURAL AND DECLARATIVE COMPONENTS

Having described both the logic programming and functional programming paradigms, we now consider the feasibility of a language which integrates some of the features of both programming paradigms. It should be noted here that both logic programming and functional programming are within a classification of programming which MacLennan refers to as "value-oriented" programming [Ref. 22]. He includes equational programming [Ref. 23] and relational programming as well [Ref. 24], but here we consider equational programming a more restrictive form of functional programming and relational programming a form of functional programming (since a function is a relation) which can deal with multi-valued functions. The focus, then, is on the feasibility of integrating a procedural component (functional programming) and a declarative component (logic programming) within a single language.

The non-procedural aspects of logic programming make it very advantageous for stating facts (or axioms) from which knowledge can be inferred, or about which queries can be made. Yet it is unnatural to define everything declaratively. For example, most PROLOG implementations define numbers and the operations on them procedurally. In

such instances it is not only unnatural to define them declaratively, it is less efficient [Ref. 25].

By contrast, the non-declarative aspect of functional programming can make the manipulation of information in a knowledge base very tedious and inconvenient. Since the search of such a knowledge base is explicit, the programmer must define functions that perform the search or comparisons required. These contrasting aspects of both programming paradigms will be illustrated and explained in the following examples.

B. EXAMPLES OF CONTRAST

1. Declarative Versus Procedural

Consider the PROLOG program in Figure 4.1 and the LISP program in Figure 4.2. The program in figure 4.1 can be used to find out such information as the annual salary, weekly tax, etc. of an employee asserted in the database. By merely satisfying the goal

```
<-- weekly_tax(John_Doe, X)
```

the system will perform the necessary resolution, backtracking and unification to produce the weekly tax of John Doe. Similarly, the LISP program in Figure 4.2 will return that individuals weekly tax when the function

```
weekly_tax(John_Doe)
```

is called. In comparison, note that although the three clauses for weekly_tax in the logic program can be defined

by one function (with a conditional) in the functional program, both programs perform the same function and their difference is primarily syntactical.

```
weekly_salary(John_Doe,500).

weekly_salary(Jim_Jones,350).
                  .
                  .
                  .

annual_salary(X,Y) <-- weekly_salary(X,Z),
                       Y is (Z * 48).

weekly_tax(X,Y) <-- annual_salary(X,Z), Z >= 20000,
                    weekly_salary(X,B),
                    Y is (B * .06).

weekly_tax(X,Y) <-- annual_salary(X,Z), Z >= 10000,
                    Z < 20000, weekly_salary(X,B),
                    Y is (B * .04).

weekly_tax(X,Y) <-- annual_salary(X,Z), Z < 10000,
                    weekly_salary(X,B),
                    Y is (B * .02).
```

Figure 4.1 PROLOG Program

However, a query to the logic program such as

```
<-- weekly_tax(X,Y), weekly_tax(Z,Y) ,
```

which will return all pairs of employees that pay the same weekly tax, is not possible in the functional program

without defining a nested function that explicitly checks the database against a conditional expreasion and constructs a new list with the results.

```
(SETQ  weekly_salary(cons 'John_Doe 500)
       (cons 'Jim_Jones 350 ...))
DEFUN((
     (weekly_tax(Name)
       (PROG (WS)
         (SETQ WS (cdr (sassoc  Name weekly_salary)))
         (COND
           (( GEQ 20000 (TIMES  48 WS ))
                      (TIMES  .06 WS )
           (( GEQ 10000 (TIMES  48 WS ))
                      (TIMES  .04 WS )
           (T    (TIMES  .02 WS )) )) ))
```

Figure 4.2 LISP Program

2. PROLOG Use of Cut

This illustrates that the pattern matching in PROLOG, resulting from the resolution of subgoals, is an advantage during the search of the database, because the method of search is at a lower level than that of the program. However, the same PROLOG implementation of logic

programming that offers this abstraction also requires the use of the "cut" symbol. The cut symbol is a means of halting unnecessary or unwanted backtracking. Its use within the clauses of a logic program requires the programmer to be intimately familiar with the method of backtracking, or side effects may be introduced into the program. This is because the cut symbol alters the way backtracking works after its use. The effect of the cut is to remove the place markers for certain goals so that they cannot be resatisfied, and commits the system to every unification made since that clause was entered [Ref. 2: pp. 64-68].

The side effects of using the cut symbol arise from the fact that a clause may be used in a manner for which it was not intended. For instance, consider the two clauses:

```
append([],X,X) <-- "cut".
append([A|B],C,[A|D]) <-- append(B,C,D).
```

where the cut prevents unnecessary backtracking. When resolving goals like

```
<-- append([a,b,c],[d,e],X)
```

or

```
<-- append([a,b,c],X,Y)
```

the cut works as intended and is appropriate from an efficiency standpoint. However, if the goal

```
<-- append(X,Y,[a,b,c])
```

is resolved, it would be matched and unified with the first

clause yielding X = [] and Y = [a,b,c]. The cut would prevent any further resolution and no other answers could be generated even though others existed [Ref. 2: pp. 65-66]. As the cut is placed deeper in the body of a clause, to freeze unification made to that point, the side effects become more difficult for the programmer to predict.

In an attempt to provide a means of controlling the cost of backtracking, the PROLOG implementation of logic programming requires the programmer to be aware of the underlying backtracking mechanism, introduces possible side effects, and negates the advantage gained by keeping the resolution mechanism at a lower level than that of the programmer.

C. INTEGRATION

The previous examples help to illustrate certain problems and inadequacies that result from either a strictly procedural approach to programming, or from a declarative approach interspersed with procedural features for efficient control. The PROLOG implementation of logic programming is a somewhat integrated approach, though to a very small degree, and the major problems with that approach have been described. The existence, and utility, of PROLOG gives some credence, then, to the feasibility of an integrated language. However, the problems with PROLOG seem to stem from the features of the language which are somewhat foreign

to Kowalski's concepts (described in chapter 2) of separation of logic and control, and LUSH resolution of Horn clauses.

In keeping with these concepts, a procedural component, namely functional programming, can provide the control characteristics for an effective and efficient declarative component, as well as provide a means for representing non-declarative knowledge.

1. Procedural Call From Declarative Component

To illustrate this notion, consider first the ability to call the procedural component from the declarative component. For example, the logic program in Figure 4.1 used three clauses to define weekly_tax. The

```
weekly_tax(X,Y) <-- weekly_salary(X,B),
                    annual_salary(X,Z),
                    Y is #(COND
                      ((GEQ 20000 Z)
                           (TIMES .06  B)
                        ((GEQ 10000 Z)
                             (TIMES .04  B)
                        (T  (TIMES .02  B) )) ))
```

Figure 4.3 Call to Procedural Component

advantages gained by backtracking within a single clause become cumbersome, tedious, and expensive when backtracking must occur through several clauses that have the same head. In an integrated approach, the ability to define weekly_tax (Figure 4.3) with a procedural call (denoted by the "#" symbol), which performs the conditional control, allows the instantiated values of the variables Z and B, unified through resolution, to be used in the function to instantiate the variable Y. The requirement for three clauses with the same head has been reduced to one clause where resolution is no longer needlessly replicated.

In such an example, note that the functional call from the declarative component merely returns the value of the evaluated function. Such a function can be evaluated by theprocedural interpreter and is regarded as a standard functional expression.

2. Declarative Call From Procedural Component

A call to the declarative component from the procedural component, on the other hand, requires a rethinking of the resolution process. For example, Figure 4.4 illustrates such a call in an attempt to use the advantages of resolution, instead of explicitly defining a function, to search the knowledge base and perform the unifications which provide a solution. But in this case the call to the declarative component cannot merely return the result of an evaluated expression. In this simple example,

```
(defun  append ( L M )
  (cond
   (( null L) M )
     (T (cons (car L) (append (cdr L) M )) )) )


append ( #(weekly_Tax(X,30), weekly_tax(Y,30)),
           '(John_Smith) )
```

Figure 4.4 Call to Declarative Component

the procedural component expects the declarative call to return a list of employees which pay 30 dollars in weekly tax. However, suppose the uninstantiated variable Z was in place of the 30. The resulting list of employees would be dependent upon the various instantiations of Z, and without knowing which employees were associated with those values of Z, the resulting list would be meaningless. What about a 3-tuple list which yields the X, Y, and Z instantiations of every solution? Such a list could become a rather large list of permutations with, possibly, redundant information. Here the results to be returned are more complicated and require modification to the resolution mechanism within the declarative interpreter.

Such modification, discussed in the following chapter, is based upon two important observations that arise from

the previous example:

- Queries (calls to the declarative component) must be qualified. Qualifications such as 'all', 'any', etc. must be made explicitly in the query.

- Unification must be based upon the given context (or environment) in which the declarative call is made. A context is a list of unifications (or bindings) which provide a constraint upon the resolution of the declarative clause.

These observations form the foundation upon which the features of a truly integrated language can be described.

V. FEATURES OF AN INTEGRATED LANGUAGE

A. SYNTAX ISSUES DEFERRED

Whether the syntax of an integrated language should be uniform (either functionally based or logically based), or whether it should be mixed, is an issue that will be deferred at this point. The intention is to describe the important features of an integrated language and discuss the modifications required of the resolution process within the declarative component. In keeping with the examples of the previous chapter, the use of PROLOG and LISP syntax can adequately represent the points to be made, and the mixed syntax will better illustrate the transfer of control from procedural interpretation to declarative, and vice versa.

With this transfer in mind, the "#" symbol will represent the transfer of control from one component to the other. The results from a procedural call (function application) within the declarative component must be the value of the evaluated function, and will be used to instantiate (or unify) a variable within the clause. The results from a declarative call (resolution process) within the procedural component, on the other hand, must return a list of solutions to the query. Each solution is itself a list of variable bindings that provide a solution to the given query.

B. QUALIFICATION OF A QUERY

As described in the previous chapter, a call to the declarative component must be qualified to ensure that expected, and meaningful, results are returned to the procedural component. In general, the programmer may require several different types of results from the declarative component. In some instances the programmer may require a list of all possible solutions to a given query. But in other instances, the programmer might require only a limited number (or even one) of all the possible solutions.

Based upon the functions described in Robinson's LOGLISP [Ref. 26] these qualifiers can be represented by the following:

ALL - returns, as a result of the declarative call, a list of all tuples which satisfy the query within the constraints of the current context (details of resolution within a context are in a later section).

ANY K - returns a list with no more than K of the tuples returned by ALL.

With these two qualifiers as the foundation, the programmer may use the functional component of the language to define, for convenience, functions which perform special, or redundant, cases of the basic qualifiers. For example, the qualification ANY 1 to the given goal statement, will return a list containing the single list of variable bindings that provides one solution. A function THE, which

will return those bindings in a single list, and may be defined by

THE (Q) = #ANY 1 (Q).

C. QUERIES AND THEIR CONTEXT

Associated with each query, or goal statement to be resolved, is an implicit context within which certain constraints are placed upon the resolution process. These constraints are variables which are already bound to values. If variables that are part of the goal statement are already bound, then those variables cannot be re-instantiated during the resolution process. For those variables that are not defined in the current context of the query, they are considered free variables and can be instantiated (or bound) during the resolution process.

1. Context Description

The context associated with each query contains all bound variables, of local scope, which may be bound to terms or to other variables. The fact that variables may be bound to other variables makes the unification somewhat more complicated, but is necessary to allow the resolution of the goal to progress as intended. It should be noted, however, that such indirect binding must eventually terminate with a binding to a term within that context.

The bindings within a given context can be denoted in a manner that lends itself to LUSH resolution (see

Chapter 2), which will simplify the checking of variables within that context. For example,

X <-- Y

Y <-- a

Z <-- b

represents the context (variables in uppercase, terms in lowercase) of a given query Q, within which X is bound to Y, Y, in turn, is bound to "a", and Z is bound to "b". Therefore, a query such as

P(T,X,V)

would have two free variables, T and V, and the variable X would be bound to the term "a" during each resolution process in the search for a solution.

2. Context Algorithm

For a given query Q and its associated context C, the constraints placed upon the resolution of Q are represented by the variables that are already bound. The following algorithm is concerned with a query of the form

$$Q(P_1,P_2, \ldots ,P_m)$$

where

$$P_i(X_1,X_2, \ldots ,X_n)$$

represents each predicate, and C is the context of the query.

Algorithm:

repeat

for (each variable X_j of every predicate P_i, in Q)

resolve $C(X_j)$ (perform LUSH resolution on)

(database C)

if $C(X_j)$ = nil

then X_j is a free variable and do nothing

else instantiate X_j to term t_k, returned from

resolution of $C(X_j)$, within the query Q

place the binding tuple in the list S_0 that

will contain all such bindings

until (all variables are checked).

Once these constraints have been placed upon Q, the query can then be resolved (subject to the modifications described below).

D. MODIFICATION TO THE RESOLUTION PROCESS

Most PROLOG implementations provide for the resolution of goal statements based upon the LUSH resolution described in Chapter 2. When a programmer makes a query to the knowledge base and a solution is provided, the programmer may then induce a failure (usually by hitting return) which invokes backtracking in search of another distinct solution. For our integration purposes, the qualifiers ALL and ANY K determines at the outset whether one, more, or all solutions are required.

From the above context algorithm, the constraints placed upon the query Q are defined in the list of bindings S_0. This list contains the instantiated variables of Q (if any), and have been so instantiated in Q. The query Q may now be resolved by the declarative component, the PROLOG interpreter for this example, until a solution is found or a failure is obtained. Given a solution is found, let the associated list of all solutions be denoted by

$$S = (S_1, S_2, \ldots, S_k, \ldots, S_m)$$

where S_1 is the first solution obtained in the resolution process.

In constructing such a solution, the following algorithm represents the modification to the resolution mechanism of the declarative interpreter that will allow its construction. The algorithm modifies the basic declarative interpreter such that the entire results of the qualified query are returned as a list of bindings which satisfy that query. Once the declarative resolution mechanism returns a failure, there are no more solutions to be obtained, and the search is halted. If it is the first attempt at a solution, then the empty list is returned, otherwise the list of solutions to that point are returned.

Algorithm:

Given the qualifier to the query, the query Q, and the constraint bindings in C,

```
First i,
repeat
  resolve Q
  if FAIL  (from resolution of Q)
    then  if i = 1
            then  S_i = ()
                  place S_i in S
          HALT;
    else  (have a solution)
      place binding tuples of solution in a list
                  (returned from resolution of Q)
      append this list to S_0
                  (to include initialization bindings)
      let S_i denote new list
      place S_i in S
      induce FAILURE  (standard PROLOG mechanism to )
                      (search for another solution  )
    if qualifier = ANY K   (check qualifier)
      then  if i >= K
              then  HALT;
    Next i
until HALT
return S.
```

Notice that the binding constraints placed upon the initial query must be explicitly included with the solution bindings resulting from the resolution of Q. These bindings

were saved in S_0 when the constraint instantiations were made in Q. They must be returned as part of solutions in S since they provide variable bindings that are part of the solution.

Although there is no explicit check for the ALL qualifier in the algorithm, it is felt that its syntactic inclusion, as part of the qualification to the query, will provide more regularity and structure to the integration interface.

E. AN INTEGRATION EXAMPLE

To illustrate the extensibility offered the programmer from the functional component, consider the function in Figure 5.1. This function takes the list of solutions obtained from the declarative call, represented by

$$S = (S_1 \; S_2 \; .... \; S_m)$$

where

$$S_i = ((X_1 \; t_1) \; (X_2 \; t_2) \; ... \; (X_n \; t_n)),$$

and allows the programmer to specify which variables within the declarative call will be returned as meaningful results. This general function provides the programmer considerable flexibility in utilizing or manipulating the results obtained from the declarative call.

For example, consider a knowledge base of facts concerning the armed forces of various countries, their mobilization status, geographical relationships, etc.

```
Return_list =  ((NLAMBDA  L)
                (PROG ( S VARLIST RESULTS )
                 (SETQ S (EVAL (LAST L)))
                 (SETQ VARLIST (LDIFFERENCE L
                                            (LAST L)))
                 (SETQ RESULTS
                   (MAPCAR S 'LAMBDA (Si)
                     (MAPCAR VARLIST 'LAMBDA (X)
                             (ASSOC X Si )) )) ))
```

Figure 5.1 Function Definition

Then a question of the form

"Which Warsaw Pact countries have exercised armored divisions within the past six months, and what divisions were they?"

could be handled by the following function (which makes a call to the declarative component):

```
Return_list (X Y (#ALL (country(X), warsaw_pact(X),
                        armored_division(X,Y),
                        mobilize(Y,D), D > Z )))
```

Here we assume that the variable Z, representing a Julian date, has been bound outside the function call (perhaps based upon a previous query regarding information within the

last six months). Therefore, the variable Z is in the current context C, is instantiated in the query, and saved in S_0, based upon the context algorithm described above.

Turning to the function in Figure 5.1, the solutions returned in S (based upon the modified resolution algorithm above) are, in effect, associated with the variables explicitly listed as an argument to Return_list, and only these values are returned. In this case the variables X and Y are listed, and may be exemplified by results such as

((Poland Fifth_armored) (Poland Seventh_armored)
(Hungary Second_armored) (DDR Second_armored)).

Other queries can be made to the declarative component based upon the solutions provided by the previous results. For instance, a follow-on question like

"Where is the Second armored division of the DDR currently located?"

could be resolved by first instantiating variables X and Y outside the query (possibly using SETQ) and using the function Return_list again with a different call to the declarative component. Therefore, the function call

Return_list (W (#ANY 1 (armored_division(X,Y),
current_location(Y,W))))

would return the current location (given such information is in the knowledge base) of the instantiated armored division.

F. SUMMARY

The previous examples illustrate the notion of a procedural component providing the control for logical relationships in the declarative component of an integrated language. They demonstrate the concept of resolution within a context, described above, and illustrate the flexibility provided the programmer, in that user defined functions can be created to best utilize or manipulate the results provided by a query to the declarative component.

By strict enforcement of the separation of logic and control, the use of the "cut" symbol can be eliminated. Its use in PROLOG is based upon the fact that the programmer is required to provide control mechanisms within the logical relationships that are created. A logical relationship that is so complex that a cut is used by the programmer (to effectively save information to that point by halting the backtracking mechanism) must be simplified in a way that makes each logical relationship a separate entity. Therefore, the programmer still has the burden of understanding the manner and method with which logical relationships are defined in the declarative component, but, more importantly, the requirement to understand the low-level details of backtracking is removed.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The previous chapter described the initial features to be considered in the integration of logic programming and functional programming. Presented has been an argument that an integrated language better supports Kowalski's notion of separation of logic and control [Ref. 8]. This argument has been based upon the idea that declarative sorts of knowledge (facts and logical relationships) should be expressed in a declarative way, and that procedural sorts of knowledge (manipulation, control, and utilization of data) should be expressed in a procedural way.

Toward this end, the declarative component of an integrated language establishes a knowledge base of facts (or assertions) as well as rules for associating those facts, determining logical relationships among them, or even inferring new knowledge and relationships. The procedural component, then, is the interactive tool for explicitly controlling those logical relationships and the knowledge base of facts upon which they are built.

The explicit control afforded by the procedural component has eliminated redundant and unnecessary backtracking. Since multiple rules are no longer required to define a single logical relationship, redundant

backtracking through the bodies of several clauses with the same head is avoided. This allows the programmer to associate one clause with one logical relationship, providing clearer understanding and easier modification and maintenance.

The necessary control for utilizing and manipulating results obtained from a query to the declarative knowledge base is provided by the programmer. This control, however, is no longer at the low level required when using the "cut" symbol. The control is now concerned with logical relationships and avoids the side effects resulting from the use of the "cut".

Additionally, the programmer is no longer concerned with explicitly defining the search of a knowledge base of assertions and rules. By using the procedural component to manipulate the results obtained from a query to the declarative component, the programmer can focus on higher-level issues of interrelationships among the results of such a query, not on the lower-level details of how that search was performed.

All of these conclusions support the ideas of abstraction, higher-level focus, and information hiding, discussed in Chapters 1 through 4. The argument for an integrated language, based upon the features described in Chapter 5, is conceptually sound, and has further supported the idea that representing varied forms of

knowledge in a strictly procedural, or strictly declarative, manner forces the programmer to contort the representation of one form of knowledge to fit its expression in another form.

B. RECOMMENDATIONS

Having provided a conceptual framework for the design of an integrated language, future emphasis should be placed upon more detailed design, and eventual implementation, of each of the procedural and declarative components of the language. A decision must be made regarding the choice of syntax (uniform or mixed) of the language, and the detailed features of each component must be based upon that decision. For instance, the choice of a functionally-based uniform syntax would require a redesign of the manner in which logical assertions and relationships are represented and interpreted. Such a redesign, however, may greatly simplify the integration interface described in the previous chapter.

Additionally, emphasis must be placed upon issues which were of concern regarding each programming paradigm in and of itself. Such issues are efficiency considerations and parallelism. With regard to efficiency, both the functional programming language and the logic programming languages are inherently slow without adequate hardware support. This slowness is a result of recursion in the functional language

and searching in the logic language. Having an integrated language with both features emphasizes the necessity for the hardware support for parallel execution.

Fortunately, both functional programming and logic programming support the notion of parallel execution, and with adequate hardware support, an integrated language could provide the best features of a functionally-based procedural component, as well as the best features of a logically-based declarative component, and that is sufficiently efficient to provide timely calculations and results.

LIST OF REFERENCES

1. Whorf, Benjamin L., Language, Thought, and Reality, MIT Press, 1956.

2. Clockain, W. F. and Mellish, C. S., Programming in Prolog, Springer-Verlag, 1981.

3. Backus, John, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs," Communications of the ACM, Vol. 21, No.8, pp. 614-640, August 1978.

4. Bartimo, J., "SMALLTALK with Alan Kay," Information World, Vol. 6, No. 24, June 1984.

5. MacLennan, B. J., Principles of Programming Languages: Design, Evaluation, and Implementation, Holt, Rhinehart, and Winston, 1983.

6. Booch, Grady, Software Engineering with Ada, Benjamin Cummings Publication Co., 1983.

7. McDermott, Drew, "The Prolog Phenomenon," SIGART Newsletter, pp. 16-20, July 1980.

8. Kowalski, Robert, "Algorithm = Logic + Control," Communications of the ACM, Vol. 22, No. 7, pp. 424-436, July 1979.

9. Kowalski, Robert, "Predicate Logic as a Programming Language," Information Processing 74, pp. 569-574, North-Holland, 1974.

10. Van Emden, M. H. and Kowalski, Robert, "The Semantics of Predicate Logic as a Programming Language," Journal of the ACM, Vol. 23, No. 6, pp. 733-742, October 1976.

11. MIT Technical Report 258, Description and Theoretical Analysis (using shemata) of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot, by Carl Hewitt, 1972.

12. Smith, Bruce, "Logic Programming on an FFP Machine," IEEE International Symposium on Logic Programming, pp. 177-185, February 1984.

13. Martelli, A. and Montanari, U. "An Efficient Unification Algorithm," ACM Transactions on Programming Languages and Systems, Vol. 4, pp. 258-282, 1982.

14. Paterson, M. S. and Wegman, M. N., "Linear Unification," Journal of Computer and Systems Sciences, Vol. 16, pp. 158-167, 1978.

15. Hill, R., "LUSH Resolution and its Completeness," DCL Memo 78, University of Edinburgh, 1974.

16. Van Emden, M. H., "Programming with Resolution Logic," Machine Intelligence 8, Ellis Horwood, 1977.

17. Barr, A. and Feigenbaum, E. A., The Handbook of Artificial Intelligence, Volume 1, Heuristic Press, pp. 170-179, 1981.

18. Rich, Elaine, Artificial Intelligence, McGraw-Hill, 1983.

19. Winston, Patrick Henry, Artificial Intelligence, second edition, Addison-Wesley, 1984.

20. MacLennan, B. J., Functional Programming Methodology: Theory and Practice, (tentative title), to be published by Addison-Wesley.

21. Basili, V. R. and Noonan, R. E., "A Comparison of the Axiomatic and Functional Models of Structured Programming," IEEE Transactions on Software Engineering, Vol. 6, No. 5, September 1980.

22. Naval Postgraduate School Technical Report 52-84-004, A Simple Software Environment Based on Objects and Relations, by MacLennan, B. J., 1984.

23. Hoffman, C. M. and O'Donnell, M. J., "Programming with Equations," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 1, pp. 83-110, 1982.

24. MacLennan, B. J., "Introduction to Relational Programming," Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, ACM Order No. 556810, 1981.

25. Barbuti, R., Bellia, M., Levi, G., and Martelli, M., "On the Integration of Logic Programming and Functional Programming," IEEE International Symposium on Logic Programming, IEEE Computer Society Press, pp. 160-167, 1984.

26. Robinson, J. A. and Sibert, E. E., "LOGLISP: an alternative to PROLOG," Machine Intelligence 10, Ellis Horwood, pp. 399-419, 1979.

27. McGrath, Thomas R., The Enhancement of Concurrent Processing Through Functional Programming Languages, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

28. Hoare, C. A. R., "An Axiomatic Basis for Computing Programs," Communications of the ACM, Vol. 12, pp. 576-583, October 1969.

29. Mills, H. D., "The New Math of Computer Programming," Communications of the ACM, Vol. 18, pp. 627-645, January 1975.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandra, Virginia 22304-6145	2
2.	Dudley Knox Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5100	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5100	1
4.	Office of Research Administration Code 012A Naval Postgraduate School Monterey, California 93943-5100	1
5.	Computer Technologies Curricular Office Code 37 Naval Postgraduate School Monterey, California 93943-5100	1
6.	Bruce J. MacLennan Code 52Ml Naval Postgraduate School Monterey, California 93943-5100	1
7.	Captain Bradford Mercer, USAF Code 52Zi Naval Postgraduate School Monterey, California 93943-5100	1
8.	Randy E. Rhodes 4720 Marlborough Drive Virginia Beach, Virginia 23464	3

21546

Thesis
R3736 Rhodes
c.2 On the integration of
logic programming and
functional programming.